Certifying 3-Edge-Connectivity

Kurt Mehlhorn Adrian Neumann Jens M. Schmidt

November 29, 2012

Abstract

We present a linear time certifying algorithm that tests graphs for 3-edge-connectivity. If the input graph is not 3-edge-connected, the algorithm returns a 2-edge-cut. If the input graph is 3-edge-connected, the algorithm returns a construction sequence that constructs the graph starting from the graph with two nodes and three parallel edges using only operations that preserve 3-edge-connectivity. No previous algorithm returned a certificate in the case of a 3-edge-connected input graph.

1 Introduction

Many graph algorithms answer complex yes-no questions such as "Is this graph planar?" or "Is this graph k-vertex-connected?". These algorithms are not only nontrivial to implement, it is also difficult to test their implementations extensively, as usually only small test sets are available. It is hence possible that bugs persist unrecognized for a long time. An example is the implementation of the linear time planarity test of Hopcroft and Tarjan [HT74] in LEDA [MNU99]. A bug in the implementation was discovered only after two years of extensive use.

Certifying algorithms [MMNS11] approach this problem by computing an additional *certificate* that proves the correctness of the answer. This may, e.g., be either a 2-coloring or an odd cycle for testing bipartiteness, or a planar embedding or a Kuratowski subgraph for testing planarity. Certifying algorithms are designed such that checking the correctness of the certificate is substantially simpler than solving the original problem. Ideally, checking the correctness is so simple that the implementation of the checking routine (the *checker*) allows for a formal verification. In that case, the solution of *every instance* is *correct by a formal proof* [ABMR11].

Our main result is a linear time certifying algorithm for 3-edge-connectivity. We make use of a construction due to Mader that characterizes 3-edge-connected graphs and show how to compute this construction in linear time. If the input graph is not 3-edge-connected, a 2-edge-cut is computed. Previous algorithms [GI91, NI92, TWO92, Tsi07, Tsi09] for deciding 3-edge-connectivity are not certifying; they deliver a 2-edge-cut for graphs that are not 3-edge-connected but no certificate in the yes-case.

Our algorithm uses the concept of a *chain decomposition* of a graph introduced in [Sch10]. A chain decomposition is a special ear decomposition [Lov85]. They are

used in [Sch12b] as a common and simple framework for certifying 1- and 2-vertex, as well as 2-edge-connectivity. Further, [Sch12a] uses them for certifying 3-vertex-connectivity (an implementation is available in [Neu11]). Chain decompositions are an example of *path-based* algorithms (see, e.g., Gabow [Gab00]), which use only the simple structure of certain paths in a DFS-tree to compute connectivity information about the graph.

We use chain decompositions to certify 3-edge-connectivity in linear time. Thus, chain decompositions form a common framework for certifying k-vertex- and k-edge-connectivity for $k \leq 3$ in linear time. We use many techniques from [Sch12a], but in a simpler form. Hence our paper may also be used as a gentle introduction to the 3-vertex-connectivity algorithm.

2 Related Work

Deciding 3-edge-connectivity is a well researched problem, with many different linear time solutions known [GI91, NI92, TWO92, Tsi07, Tsi09]. None of them is certifying.

The paper [MMNS11] is a recent survey on certifying algorithms. For a linear time certifying algorithm for 3-vertex-connectivity, see [Sch12a, Neu11]. For general k, there is a certifying algorithm for k-vertex connectivity in [LLW88] with running time $O(n^{2.5} + nk^{2.5})$ and error probability at most $1/n$. There is a non-certifying algorithm [Kar00] for deciding k-edge-connectivity in time $O(m \log^3 n)$ with high probability.

In [GI91], a graph transformation is described that transforms a graph G into a graph G' such that G is 3-edge-connected if and only if G' is 3-vertex-connected. However, the certifying 3-vertex-connectivity algorithm from [Sch12a, Neu11] is much more complex than the algorithm given here. Moreover, we were unable to find an elegant method for transforming the certificate obtained for the 3-vertex-connectivity of G' into a certificate for 3-edge-connectivity of G.

3 Preliminaries

We consider finite undirected simple graphs G with $n = |V(G)|$ vertices and $m = |E(G)|$ edges and use standard graph-theoretic terminology from [BM08], unless stated otherwise.

A set of edges that leaves a disconnected graph upon deletion is called *edge cut*. For $k \geq 1$, let a graph G be *k-edge-connected* if $n > k$ and there is no edge cut X in G with $|X| < k$. Vertex connectivity is defined analogously.

Let $v \to_G w$ denote a path P from vertex v to vertex w in G and let $s(P) = v$ and $t(P) = w$ be the source and target vertex of P, respectively (as G is undirected, the direction of P is given by $s(P)$ and $t(P)$). Every vertex in $P \setminus \{s(P), t(P)\}$ is called an *inner vertex* of P.

Let T be an undirected tree rooted at vertex r. For two vertices x and y in T, let x be an *ancestor* of y and y be a *descendant* of x if $x \in V(r \to_T y)$. If additionally $x \neq y$, x is a *proper* ancestor and y is a *proper* descendant. We write $x < y$ if x is an ancestor

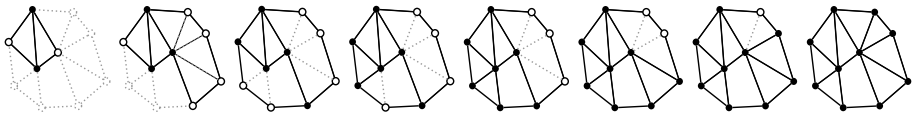

Figure 1: Construction of a 3-edge-connected graph using Mader-paths. Real vertices are depicted in black. The black edges exist already, while dotted grey vertices and edges do not exist yet.

of y. The parent $p(v)$ of a vertex v is its immediate ancestor. The parent function is undefined for r. Let K_2^m be the graph on 2 vertices that contains exactly m parallel edges and no self-loops. A decomposition of G is a set of subgraphs of G whose edge-sets partition $E(G)$.

Let *subdividing an edge uv* of a graph G be the operation that replaces uv with a path uzv, where z was not previously in G. Then a subdivision G' of a graph G is a graph obtained by a series of subdivisions. We call paths in G' that correspond to edges in G *links*. The *real* vertices of a subdivision are the vertices with degree at least three.

In [Mad78], Mader shows that all 3-edge-connected graphs can be constructed using a small set of operations starting on a K_2^3.

Theorem 1 (Mader [Mad78])**.** *Any 3-edge-connected graph can be constructed from a K_2^3 using the following three operations:*

- *Adding an edge (possibly parallel or a loop).*
- *Subdividing an edge xy and connecting the new vertex to any existing vertex.*
- *Subdividing two edges wx, yz and connecting the two new vertices.*

When we talk about subdivisions of 3-edge-connected graphs, the analogous operations add links. Edges are no longer subdivided as in the last two operations of Theorem 1; instead, links are added that end at inner vertices of other links, i.e. the vertex that would have been created by the subdivision is an inner vertex of a path that was added previously. We call the paths in this process *Mader-paths*. For an example construction, see Figure 1.

Proposition 1. *Let G be a subdivision of a 3-edge-connected graph and let C be a simple path disjoint from G except for its endpoints. If the endpoints of C are not both interior points of the same link of G, then C is a Mader-path for G.*

Proof. The only forbidden Mader-operation is subdividing the same link twice. □

In this paper we show how to find a Mader construction sequence for a 3-edge-connected graph in linear time.

4 Chain Decompositions

We use a very simple decomposition of graphs into cycles and paths. The decomposition unifies existing linear-time tests on 2-vertex- and 2-edge-connectivity [Sch12b]

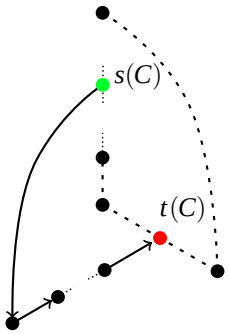

Figure 2: A chain C (solid black) with the green vertex being $s(C)$, the red vertex being $t(C)$ and the dashed chain being $p(C)$.

and is also the base structure for certifying 3-vertex-connectivity efficiently. In this paper we show that it can also be used to find a construction sequence for a 3-edge connected graph. We define the decomposition algorithmically; a similar procedure that serves for the computation of low-points can be found in [Ram93].

Let G be a simple but not necessarily connected graph and let T be a depth-first search forest of G. The DFS assigns a depth-first index (DFI) to every vertex. For every backedge e, $s(e)$ and $t(e)$ are the two end vertices of e such that $s(e)$ is a proper ancestor of $t(e)$ in T.

We decompose G into a set $\mathscr{C} = \{C_1, \ldots, C_{|\mathscr{C}|}\}$ of cycles and paths, called *chains*, by applying the following procedure for each vertex v in ascending DFI-order: Let T' be the tree in the DFS-forest T that contains v and let r be the root of T'. For every backedge vw with $s(vw) = v$, we traverse the path $w \to_{T'} r$ until a vertex x is encountered that was visited before. The traversed subgraph $vw \cup (w \to_{T'} x)$ forms a new *chain* C_i with $s(C_i) = v$ and $t(C_i) = x$. Since every backedge defines one chain, there are precisely $m - n + 1$ chains.

If x is not r, we set the *parent* of C_i as $p(C_i) = C_p$. We say a vertex v *belongs to* a chain C if $vp(v) \in C$; by construction a vertex belongs to a unique chain. We define an order on the chains by setting $C_i < C_j$ if $i < j$. See Figure 2.

We call $\mathscr{C}$ a *chain decomposition*. Clearly, C can be computed in time $O(n+m)$. We will use the following lemma.

Lemma 1 ([Sch12b])**.** *Let $\mathscr{C}$ be a chain decomposition of a simple graph G. Then*

- *G is 2-edge-connected if and only if G is connected and the chains in $\mathscr{C}$ partition $E(G)$.*
- *G is 2-connected if and only if it has minimum degree at least two and C_1 is the only cycle in $\mathscr{C}$.*

5 The Algorithm

An implementation in Python is publicly available at `https://github.com/adrianN/edge-connectivity`. The overall structure of the algorithm is given in Algorithm 1.

5.1 Computing the Certificate

Algorithm 1 Certifying algorithm for 3-edge connectivity

```
procedure CONNECTIVITY(G=(V,E))
    chains = MODIFIED_CHAIN_DECOMPOSITION(G)
    for chain in chains do
        for c in chain.children[type 2] do
            ADD(c)
        end for
        segments = []
        for c in chain.children[type 1] do
            segments.append(Segment(c))
        end for
        for c in chain.type3 do
            segment = MAKE_SEGMENT(c)
            if segment ∩ chain.children[type 1] = ∅ then
                ADD_TYPE3(c)
            end if
        end for
        ORDER_AND_ADD(segments)
    end for
end procedure
```

We show how to find a Mader sequence in linear time. The algorithm begins by computing a chain decomposition as described above and then modifying it slightly. In this phase we also check that the graph is 2-edge-connected using Lemma 1 and that it has minimum degree at least three. We will also use the following necessary condition for 3-edge-connectivity.

Proposition 2. *Let T be a DFS tree starting at r for a graph G. If G is 3-edge-connected, then the subtree of every child of r must be connected to r by at least two backedges.*

Using the chain decomposition, we can identify a K_2^3 subdivision in the graph as follows. The first chain, C_1, of the chain decomposition forms a cycle. By Proposition 2, there must be a chain that starts at r and has C_1 as a parent, otherwise the graph is not 3-edge connected. W.l.o.g. we can assume that C_2 is this chain and $t(C_2) = x$. We modify the first chain of the chain decomposition by setting $C_0 = x \to_T r$ and $C_1 = r \to_{C_1} x$. Then $C_0 \cup C_1 \cup C_2$ form an initial K_2^3 subdivision. Note that we only do this transformation once, later chains might be cycles if the graph is not 2-vertex-connected.

Note that, except for r, every vertex belongs to the first chain that contains it. Starting with $G_3 = C_0 \cup C_1 \cup C_2$, our goal is now to find either an order π on the remaining chains such that

$$G_k = \bigcup_{0 \le j < k} C_{\pi(j)}$$

and $C_{\pi(k)}$ is a Mader-path for G_k for every $k \in \{3, \ldots, m-n+1\}$ or to find a separating pair of edges for G if no such order exists. As the G_i are constructed using Mader-paths, Theorem 1 implies that they are subdivisions of 3-edge-connected graphs. We say that we *add* a chain C_i in step k if it is a Mader-path with respect to G_k and $\pi(k) = i$, and that a chain C_i is *part of the current graph* in step k if $C_i \subseteq G_k$.

We classify the chains $\{C_1, \ldots C_{m-n+1}\}$ into three types. Let C be a chain with parent $C' = p(C)$. By construction, $s(C)$ is a (not necessarily proper) descendant of $s(C')$. We classify C according to the location of $s(C)$. There are three cases.

- If $s(C)$ belongs to C', C has *Type 1*. Observe that $t(C) \to_T s(C)$ is contained in C' for type 1 chains.
- If $s(C) = s(C')$, the chain is of *Type 2*. We also set the type of the chains starting at r and having parent C_0 to two, even though $s(C_0)$ is not r.
- Otherwise, i.e. if $s(C)$ is an ancestor of $t(C')$ and a proper descendant of $s(C')$, C is of *Type 3*.

Let G_c be a union of chains. G_c is *upwards-closed* if, for any vertex $v \in G_c$, the edge $vp(v)$ is also part of G_c.

Lemma 2. *Let G_c be a upwards-closed union of chains that contains C_0, C_1, and C_2. Then any type two or three child of a chain in G_c is a Mader-path with respect to G_c.*

Proof. Let C be a chain in G_c. If $C = C_0$, $s(C)$ is real as C_1 and C_2 are in G_c. Otherwise, by the upwards-closedness of G_c and the construction of chains, $s(C)$ is real as the edge $s(C)p(s(C))$ is in G_c and belongs to a chain different from C. Further, all children of C are ears with respect to G_c, that is, they are disjoint from G_c save for their endpoints.

We apply Proposition 1. Let C' be a child of C. If C' has type two, $s(C') = s(C)$ and hence C' is a Mader-path. If C' has type three, we have $s(C) < s(C') < t(C) < t(C')$. Since $t(C)$ is real, $t(C') \to_T s(C')$ is not a link. As G_c is a union of chains, no other path $t(C') \to_{G_c} s(C')$ can be a link either. Hence C' is a Mader-path. □

We operate in phases. In phase i, $i \in [0, |\mathscr{C}|]$, we first add all type two children of C_i and then all chains contained in *segments* with respect to C_i. The segments with respect to C_i are sets of descendants of C_i. They are defined by the following rules.

- Every type one child of C_i belongs to some segment.
- Every type three chain C with $s(C) \in C_i$ belongs to some segment.
- If C belongs to some segment and $p(C)$ does not belong to the current graph (i.e. the graph after adding all type two children of C_i) then $p(C)$ belongs to the same segment with respect to C_i.

- That is all, i.e., only chains forced by one of the rules belong to a segment with respect to C_i and segments are disjoint except if merged by the third rule.

If these chains cannot be added, we will exhibit a separation pair.

As we shall see in Lemma 3, each segment is a subtree of the chain tree that contains no chains of type one, except possibly a type one child of C_i as its root. In any segment the chains belonging to it are added parent-first, i.e., a parent is added before its children. In the sequel, we give details, in particular, we discuss in what order the segments are added. We refer to C_i as the current chain of phase i.

Proposition 3. *The current chain is part of the current graph at the beginning of any phase.*

Proof. The initial current graph consists of chains C_0, C_1 and C_2. If C_i is of type one or two, it was added in the phase for its parent. If it is of type three, let $s(C_i)$ lie on C_j. Then C_i is in a segment of C_j; since $j < i$, C_i was added in the phase for C_j. □

Proposition 4. *The current graph is always upwards-closed.*

Proof. Directly from the fact that for all chains that we add, the parent is already added, and the fact that we add nothing but whole chains. □

We now come to the details of a phase. By Lemma 2, type two children of C_i can be added. We do so. Let G_c be the current graph after the addition of the type two chains. We will next discuss how to add the segments with respect to C_i. Segments correspond to subtrees of the chain tree. Only the roots of the subtrees can be of type one as we next show.

For a type three chain D_0 that is not part of the current graph and satisfies $s(D_0) \in C_i$, let the *subsegment* corresponding to D_0 be all ancestors $D_k < \cdots < D_0$ of D_0 that are contained in a segment of C_i.

Lemma 3 ([Sch11, Lemma 81])**.** *Let D be a chain of type three with $s(D) \in C_i$ and let D' be any ancestor of D such that $p(D')$ is not part of the current graph. Then D' has type two or three.*

Proof. Assume otherwise, i.e. D' has type one. Then $D' \neq D$, since D has type three. Moreover, $s(D')$ is a proper ancestor of $s(D)$ as otherwise D would be of type two. Since $D'' = p(D')$ is not part of the current graph G_c, $t(D')$ cannot be part of the current graph and hence, by upwards-closedness, $t(D')$ must be a proper descendant of $s(D)$. It follows that $s(D)$ is an inner vertex of $t(D') \to_T s(D')$.

Since D' is of type one, D'' contains the path $t(D') \to_T s(D')$ and hence $s(D) \in D''$. Thus $s(D) \notin G_c$, a contradiction. □

In particular, if D' is of type one, it is the minimal chain in the segment and a child of C_i.

Corollary 1. All chains in a segment S can be added in parent-first order if its minimal chain can be added.

Figure 3: Intervals for the blue segment with attachment points 1,2,4.

Proof. By Lemma 3 all but the minimal chain in a segment are of type two or three. Hence the claim follows from Lemma 2. □

Next we describe how we compute segments. We only store segments that contain a type one child of C_i, the others can be added immediately due to Lemma 2 and Corollary 1. We compute the segments by traversing the subsegments of all type three chains C with $s(C) \in C_i$, starting with the chain having a source vertex with minimal DFI (i.e. the minimal chain w.r.t. $<$). If the minimal chain in the subsegment is not of type one, we immediately add all chains in the subsegment. Otherwise, we mark all its chains with the minimal chain of the segment. To achieve linear runtime, we stop the construction of a subsegment once a marked chain is reached.

It remains to compute a proper ordering of the segments containing a type one chain or to exhibit a separation pair. For simplicity, we will say 'segment' instead of 'segment containing a type one chain' from now on.

For a segment S let the *attachment points* of S be all vertices in S that lie on C_i. Note that the attachment points must necessarily be endpoints of chains in S and hence adding the chains of S makes the attachment points real. Type one children C of C_i can be added if there are real vertices on $t(C) \rightarrow_T s(C)$, therefore adding a segment can make it possible to add further segments. We reduce the problem of finding an order on the segments to a problem on intervals.

W.l.o.g. assume that the vertices of C_i are numbered consecutively from 1 to $|C_i|$. For every segment S, let $a_0 \leq a_1 \leq \ldots \leq a_k$ be the set of attachment points of S. We associate the following intervals with S

$$\{[a_0, a_\ell] | 1 \leq \ell \leq k\} \cup \{[a_\ell, a_k] | 1 \leq \ell < k\},$$

and for every real vertex v on C_i we define an interval $[0, v]$. See Figure 3 for an example.

We say two intervals $[a, a'], [b, b']$ *overlap* if $a \leq b \leq a' \leq b'$. Note that overlapping is different from intersecting; an interval does not overlap intervals in which it is properly contained or which it properly contains. The overlap relation naturally defines an overlap graph for a set of intervals. For a set of segments $S_1, \ldots, S_k$, let the *reduced overlap graph* be the graph on the segments and a special vertex for the real vertices in which there is an edge between two vertices if any intervals associated with these vertices overlap.

Lemma 4. *If the reduced overlap graph H for C_i is connected, we can add all segments of C_i.*

Proof. Let $R, S_1, \ldots, S_j$ be the vertices of H in a preorder, e.g. the order they are explored by a DFS, starting at R, the vertex corresponding to the real vertices on C_i. An easy inductive argument shows that we can add all segments in this order. Let C be the minimal chain of S_1. An edge between R and S_1 implies a real vertex on the path $t(S_1) \to_T s(S_1)$, hence S_1 can be added by Corollary 1. Since the attachment points of a segment become real when it is added, a later segment S_i in the preorder can be added due to similar reasoning after its predecessors are added.

If H is connected, this adds all segments. On the other hand suppose H is not connected and let $\mathscr{S}$ be a connected component that does not contain R. Let a be the minimal attachment point of any chain in $\mathscr{S}$ and let b be the maximal one. Note that a and b are both different from $s(C_i)$ and $t(C_i)$, as these are real.

Observe that all edges incident to vertices on $b \to_T a$ belong either to C_i or to a chain in $\mathscr{S}$. They can't belong to ancestors of C_i, otherwise they would create real vertices, and they can't belong to a segment outside the connected component as they would induce an overlapping interval otherwise.

As a and b are the extremal attachment points of $\mathscr{S}$, the tree edge from a to its parent and the edge from b's predecessor on $s(C_i) \to_{C_i} t(C_i)$ to b disconnect a and b from the rest of the graph. □

It remains to show that we can find an order as required in Lemma 4, or a separating edge pair, in linear time.

A naive approach that constructs the reduced overlap graph and runs a DFS will fail, since the overlap graph can have a quadratic number of edges. However, using a method developed by Olariu and Zomaya [OZ96], we can compute a spanning forest of the graph in time linear in the number of intervals. The presentation in [OZ96] is for the PRAM and thus needlessly complicated for our purposes. A simpler explanation can be found in the appendix.

Since the number of intervals created for a chain C_i is bounded by

$$|\text{Children}_1(C_i)| + 2|\text{Type3}(C_i)| + |V_{\text{real}}(C_i)|,$$

where $\text{Children}_1(C_i)$ are the type one children of C_i, $\text{Type3}(C_i)$ are the type three chains that start on C_i, and $V_{\text{real}}(C_i)$ is the set of real vertices on C_i, the total time spent on this procedure for all chains is $O(m)$.

From the above discussion follows:

Theorem 2. *A Mader construction sequence can be found in time $O(n+m)$.*

5.2 Verifying the Certificate

The certificate is either a separating edge pair, or a sequence of Mader-paths. For a separating pair, we simply remove the two edges and verify that G is no longer connected.

Checking the Mader sequence is slightly more involved. We assume that each edge in a Mader-path is doubly linked to the corresponding edge in G. Then it is easy to check that the Mader-paths partition the edges of G.

Let G' be a copy of G. We begin by checking that the minimum degree of G' is at least three. Now we remove the Mader-paths again, in reverse order, suppressing vertices of degree two as they occur. This can create multiple edges and loops.

Let G_i' be the multi-graph before we remove the i-th path P_i. At this point, P_i must have been reduced to a single edge as inner vertices of P_i must have been reduced to degree two if P_i is an ear for G_i'. After removing the edge corresponding to P_i, it must not be the case that both endpoints are still adjacent but have degree two, as then P_i subdivided the same link twice.

Lastly we check that G_2' is a K_2^3 subdivision.

6 Conclusion

We presented a certifying linear time algorithm for 3-edge-connectivity based on chain decompositions of graphs. It is simple enough to use in a classroom setting and can serve as a gentle introduction to the certifying 3-vertex-connectivity algorithm of [Sch12a]. We also provide an implementation in Python, available at `https://github.com/adrianN/edge-connec`

There remain some open problems. Foremost, our algorithm only computes one separation pair. Is it possible to compute the 3-edge-connected components easily?

Mader's construction sequence is general enough to construct k-edge-connected graphs for any $k \geq 3$, and can thus be used in certifying algorithms for larger k. So far, though, it is unclear how to compute these more complicated construction sequences. We hope that the chain decomposition framework can be adapted to work in these cases too.

References

[ABMR11] E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. Verification of certifying computations. In *Computer Aided Verification*, pages 67–82. Springer, 2011.

[BM08] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Springer, 2008.

[Gab00] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000.

[GI91] Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, 1991.

[HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[Kar00] D. R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000.

[LLW88] N. Linial, L. Lovász, and A. Wigderson. Rubber bands, convex embeddings and graph connectivity. *Combinatorica*, 8(1):91–102, 1988.

[Lov85] L. Lovász. Computing ears and branchings in parallel. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 464–467, 1985.

[Mad78] W. Mader. A reduction method for edge-connectivity in graphs. In B. Bollobás, editor, *Advances in Graph Theory*, volume 3 of *Annals of Discrete Mathematics*, pages 145–164. 1978.

[MMNS11] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[MNU99] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[Neu11] A. Neumann. Implementation of Schmidt's algorithm for certifying triconnectivity testing. Master's thesis, Universität des Saarlandes and Graduate School of CS, Germany, 2011.

[NI92] H. Nagamochi and T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9:163–180, 1992.

[OZ96] S. Olariu and A. Y. Zomaya. A time- and cost-optimal algorithm for interlocking sets – With applications. *IEEE Trans. Parallel Distrib. Syst.*, 7(10):1009–1025, 1996.

[Ram93] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In *Synthesis of Parallel Algorithms*, pages 275–340, 1993.

[Sch10] J. M. Schmidt. Contractions, removals and certifying 3-connectivity in linear time. Tech. Report B 10-04, Freie Universität Berlin, Germany, May 2010.

[Sch11] J. M. Schmidt. *Structure and Constructions of 3-Connected Graphs*. PhD thesis, Freie Universität Berlin, Germany, 2011.

[Sch12a] J. M. Schmidt. Certifying 3-connectivity in linear time. In *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, to appear (2012).

[Sch12b] Jens M. Schmidt. *A Simple Test on 2-Vertex- and 2-Edge-Connectivity*, 2012. Manuscript (arxiv.org/abs/1209.0700).

[Tsi07] Y. H. Tsin. A simple 3-edge-connected component algorithm. *Theor. Comp. Sys.*, 40(2):125–142, 2007.

[Tsi09] Y. H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *J. of Discrete Algorithms*, 7(1):130–146, 2009.

[TWO92] S. Taoka, T. Watanabe, and K. Onaga. A linear time algorithm for computing all 3-edge-connected components of a multigraph. *IEICE Trans. Fundamentals E75*, 3:410–424, 1992.

A Computing a Spanning Subgraph of an Overlap Graph

We first assume that all endpoints are pairwise distinct. We will later show how to remove this assumption by perturbation.

For every interval $I = [a,b]$ define its set of left and right neighbors:

$$L(I) = \{I' = [a',b']; a' < a < b' < b\},$$
$$R(I) = \{I' = [a',b']; a < a' < b < b'\}.$$

If the set of left neighbors is nonempty, let the interval $I' \in L(I)$ with the rightmost right endpoint be the immediate left neighbor of I. Similarly, if the set of right neighbors is nonempty, the immediate right neighbor of I is the interval in $R(I)$ with the leftmost left endpoint.

Lemma 5. *The graph G' formed by connecting each interval to its left and right neighbor (if any) forms a spanning subgraph of the overlap graph G and has exactly the same connected components.*

Proof. Clearly, every edge of G' is also an edge of G. For the other direction, assume I and I' are overlapping intervals that are not connected in G' and for which the left endpoint of I is as small as possible. Then $a < a' < b < b'$, where $I = [a,b]$ and $I' = [a',b']$. Since $I' \in R(I)$, but I and I' are not connected, I has an immediate right neighbor $J \neq I'$. The left endpoint of J must thus be smaller than a' and the right endpoint of J must be larger than b (since I overlaps with J). If the right endpoint is smaller than b', J and I' overlap. By repeating this argument for J and I', we must reach an interval $U = [c,d]$ containing I'. Thus

$$a < c < a' < b < b' < d.$$

Starting from I' and going to left neighbors, we obtain in the same fashion an interval $U' = [c',d']$ with

$$c' < a < a' < b < d' < b'.$$

We conclude that U' and U overlap, but are not connected in G'. Since the left endpoint of U' is to the left of the left endpoint of I, this contradicts the choice of I and I'. □

It is easy to determine all immediate right neighbors by a linear time sweep over all intervals. We sort the intervals in decreasing order of left endpoint and then sweep over the intervals starting with the interval with rightmost left endpoint. We maintain a stack S of intervals, initially empty. If $I_1 = [a_1,b_1], \ldots, I_k = [a_k,b_k]$ are the intervals on the stack with I_1 being on the top of the stack, then $a_1 < a_2 < \ldots < a_k$ and $b_1 < b_2 < \ldots < b_k$, I_1 is the last interval processed, and $I_{\ell+1}$ is the immediate right neighbor of I_ℓ if I_ℓ has right neighbors. If I_ℓ does not have right neighbors, $a_{\ell+1} > b_\ell$. Let $I = [a,b]$ be the next interval to be processed. Its immediate right neighbor is the topmost interval I_ℓ on the stack with $b_\ell > b$ (if any). Hence we pop intervals I_ℓ from the stack while $b > b_\ell$ and then connect I to the topmost interval if $b > a_\ell$, and push I. The determination of immediate left neighbors is symmetric.

It remains to deal with intervals with equal endpoints. We do so by perturbation. It is easy to see that the following rules preserve the overlaps-relation and eliminate equal endpoints.

(1) if a left and a right endpoint are at the same coordinate, then the left endpoint precedes the right endpoint.
(2) if two left endpoints are equal, the one belonging to the shorter interval is smaller.
(3) if two right endpoints are equal, the one belonging to the shorter interval is larger.
(4) if two intervals are equal, one is slightly shifted to the right.

In other words, the endpoints of an interval $I_i = [a,b]$ are replaced by $((a,-1,b-a,i)$ and $(b,1,b-a,i))$ and comparisons are lexicographic. The perturbation need not be made explicitly, it can be incorporated into the sorting order and the conditions under which edges are added, as described in Algorithm 2.

Algorithm 2 Finding a spanning forest of a overlap graph

```
procedure SP(I = {[a_0, a'_0], ..., [a_ℓ, a'_ℓ]})
    stack = [ ]
    sort I lexicographically in descending order
    for [l, r] in I do
        while stack not empty and r > top(stack) right endpoint do
            pop(stack)
        end while
        if stack not empty and r ≥ top(stack) left endpoint then
            connect [l, r], top(stack)
        end if
        push(stack, [l, r])
    end for
    stack = [ ]
    sort I lexicographically in ascending order where the key for [l, r] is [r, l]
    for [l, r] in I do
        while stack not empty and l<top(stack) left endpoint do
            pop(stack)
        end while
        if stack not empty and l ≤ top(stack) right endpoint then
            connect [l, r], top(stack)
        end if
        push(stack, [l, r])
    end for
end procedure
```